Name

intro – introduction to miscellaneous library functions

Description

These functions constitute minor libraries and other miscellaneous runtime facilities. Most are available only when programming in C.

The list below includes libraries which provide device-independent plotting functions, terminal-independent screen management routines for two-dimensional nonbitmap display terminals, functions for managing data bases with inverted indexes, and sundry routines used in executing commands on remote machines. The routines `getdiskbyname`, `rcmd`, `rresvport`, `ruserok`, and `rexec` reside in the standard C runtime library "–lc". All other functions are located in separate libraries indicated in each manual entry.

Files

```
/lib/libc.a
/usr/lib/libdbm.a
/usr/lib/libtermcap.a
/usr/lib/libcurses.a
/usr/lib/lib2648.a
/usr/lib/libplot.a
```

creatediskbyname (3x)

Name

creatediskbyname – get the disk description associated with a file name

Syntax

#include <disktab.h>

struct disktab *
creatediskbyname(*name***)**
char ****name***;**

Description

The `creatediskbyname` subroutine takes the name of the character device special file representing a disk device (for example, `/dev/rra0a`) and returns a structure pointer describing its geometry information and the default disk partition tables. It obtains this information by polling the controlling disk device driver. The `creatediskbyname` subroutine returns information only for MSCP and SCSI disks.

The `<disktab.h>` file has the following form:

```
#define DISKTAB         "/etc/disktab"

struct  disktab {
      char    *d_name;          /* drive name */
      char    *d_type;          /* drive type */
      int     d_secsize;        /* sector size in bytes */
      int     d_ntracks;        /* # tracks/cylinder */
      int     d_nsectors;       /* # sectors/track */
      int     d_ncylinders;     /* # cylinders */
      int     d_rpm;            /* revolutions/minute */
      struct partition {
              int     p_size;   /* #sectors in partition */
              short   p_bsize;  /* block size in bytes */
              short   p_fsize;  /* frag size in bytes */
        } d_partitions[8];
};

struct  disktab *getdiskbyname();
struct  disktab *creatediskbyname();
```

Diagnostics

Successful completion of the `creatediskbyname` subroutine returns a pointer to a valid disktab structure. Failure of this subroutine returns a null pointer. The subroutine fails if it cannot obtain the necessary information from the device driver or disktab file.

A check is done to ensure that the disktab file exists and is readable. This check ensures that the subroutine is not being called because the disktab file was accidentally removed. If there is no disktab file, the subroutine fails.

The `creatediskbyname` subroutine also fails if it cannot determine disk geometry attributes by polling the driver. This can occur if the disk is not an MSCP or SCSI disk. In some cases where the disk consists of removable media and the media is not loaded, the driver will be unable to determine disk attributes.

Restrictions

The `creatediskbyname` subroutine returns information only for MSCP and SCSI disks.

See Also

getdiskbyname(3x), ra(4), rz(4), disktab(5)

curses (3x)

Name

curses – screen functions with optimal cursor motion

Syntax

cc [flags] files **–lcurses –ltermcap** [libraries]

Description

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the `refresh` subroutine tells the routines to make the current screen look like the new one. To initialize the routines, the routine `initscr` must be called before any of the other routines that deal with windows and screens are used. The routine `endwin` should be called before exiting.

Functions

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
crmode()	set cbreak mode
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
echo()	set echo mode
endwin()	end window modes
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getcap(name)	get terminal capability *name*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) co-ordinates
initscr()	initialize screens
insch(c)	insert a char
insertln()	insert a line
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unset raw mode
overlay(win1,win2)	overlay win1 on win2

overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*
raw()	set raw mode
refresh()	make current screen look like *stdscr*
resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchwin(win)	''change'' all of *win*
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add char to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win,c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
winsch(win,c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win,y,x)	set current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	scanf through *win*
wstandend(win)	end standout mode on *win*
wstandout(win)	start standout mode on *win*

See Also

ioctl(2), getenv(3), tty(4), termcap(3x), termcap(5)
Screen Updating and Cursor Movement Optimization: A Library Package, *ULTRIX Supplementary Documents* Vol. II:Programmer

Name

dbminit, fetch, store, delete, firstkey, nextkey – data base subroutines

Syntax

```
typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

Description

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option **–ldbm**.

*Key*s and *content*s are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by `dbminit`. At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. (An empty database is created by creating zero-length '.dir' and '.pag' files.)

Once open, the data stored under a key is accessed by `fetch` and data is placed under a key by `store`. A key (and its associated contents) is deleted by `delete`. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of `firstkey` and `nextkey`. The `firstkey` will return the first key in the database. With any key `nextkey` will return the next key in the database. This code will traverse the data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

Restrictions

The `.pag`file four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (`cp, cat, tp, tar, ar`) without filling in the holes.

The *dptr* pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. The `store` will return an error in the event that a disk block fills with inseparable data.

The `delete` does not physically reclaim file space, although it does make it available for reuse.

Return Value

Routines that return a *datum* indicate errors with a null (0) *dptr*. All functions that return an *int* indicate errors with negative values. A zero return indicates a successful completion.

Name

disassembler – disassemble a MIPS instruction and print the results

Syntax

```
int disassembler (iadr, regstyle, get_symname, get_regvalue, get_bytes, print_header)
unsigned        iadr;
int             regstyle;
char            *(*get_symname)();
int             (*get_regvalue)();
long            (*get_bytes)();
void            (*print_header)();
```

Description

The **disassembler** function disassembles and prints a MIPS machine instruction on *stdout*.

The argument is the instruction address to be disassembled. The *regstyle* parameter specifies how registers are named in the disassembly. The value is 0 if compiler names are used; otherwise, hardware names are used.

The next four arguments are function pointers, most of which give the caller some flexibility in the appearance of the disassembly. The only function that must be provided is *get_bytes*. All other functions are optional. The *get_bytes* function is called without arguments and returns the next byte or bytes to disassemble.

The *get_symname* is passed an address, which is the target of a *jal* instruction. If null is returned or if *get_symname* is null the *disassembler* prints the address; otherwise, the string name is printed as returned from *get_symname*. If *get_regvalue* is not null, it is passed a register number and returns the current contents of the specified register. The **disassembler** function prints this information along with the instruction disassembly. If *print_header* is not null, it is passed the instruction address, *iadr,* and the current instruction to be disassembled, which is the return value from *get_bytes*. The *print_header* function can use these parameters to print any desired information before the actual instruction disassembly is printed.

If *get_bytes* is null, the **disassembler** returns -1 and errno is set to EINVAL; otherwise, the number of bytes that were disassembled is returned. If the disassembled word is a jump or branch instruction, the instruction in the delay slot is also disassembled.

See Also

ldfcn(5)

Name

getdiskbyname – get disk description by its name

Syntax

#include <disktab.h>

**struct disktab *
getdiskbyname(name)
char *name;**

Description

The `getdiskbyname` subroutine takes a disk name (for example, RM03) and returns a structure describing its geometry information and the standard disk partition tables. All information obtained from the `disktab`(5) file. A separate subroutine called `creatediskbyname` dynamically generates disktab entries by obtaining disk geometry information from the controlling device driver.

<disktab.h> has the following form:

```
#define DISKTAB         "/etc/disktab"

struct  disktab {
      char    *d_name;          /* drive name */
      char    *d_type;          /* drive type */
      int     d_secsize;        /* sector size in bytes */
      int     d_ntracks;        /* # tracks/cylinder */
      int     d_nsectors;       /* # sectors/track */
      int     d_ncylinders;     /* # cylinders */
      int     d_rpm;            /* revolutions/minute */
      struct partition {
              int     p_size;   /* #sectors in partition */
              short   p_bsize;  /* block size in bytes */
              short   p_fsize;  /* frag size in bytes */
        } d_partitions[8];
};

struct  disktab *getdiskbyname();
struct  disktab *creatediskbyname();
```

See Also

creatediskbyname(3x), disktab(5)

getfsent(3x)

Name

getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent – get file system descriptor file entry

Syntax

#include <fstab.h>
#include /usr/include/sys/fs_types.h

struct fstab *getfsent()

struct fstab *getfsspec(*spec***)**
char ****spec*;**

struct fstab *getfsfile(*file***)**
char ****file*;**

struct fstab *getfstype(*type***)**
char ****type*;**

int setfsent()

int endfsent()

Description

All routines operate on the file `/etc/fstab`, which contains descriptions of the known file systems. The routine `setfsent` opens this file. The routine `getfsent` reads the next file system description within `/etc/fstab` opening the file if necessary. The `endfsent` routine closes the file.

The `getfsspec`, `getfsfile`, and `getfstype` routines sequentially scan the file `/etc/fstab` for specific file system descriptions. The `getfsspec` routine searches for a description with a matching special file name field. The routine `getfsfile` searches for a description with a matching file system path prefix field. The routine `getfstype` searches for a description with a matching file system type field.

The `getfsent`, `getfsspec`, `getfstype`, and `getfsfile` each return a pointer to a representation of the description they have matched or read. Representations are in the format of the following structure:

```
#define       FSTAB_RW      "rw"   /* read-write device       */
#define       FSTAB_RO      "ro"   /* read-only device        */
#define       FSTAB_RQ      "rq"   /* read-write with quotas */
#define       FSTAB_SW      "sw"   /* swap device             */
#define       FSTAB_XX      "xx"   /* ignore totally          */

struct fstab {
       char   *fs_spec;     /* block special device name    */
       char   *fs_file;     /* file system path prefix      */
       char   *fs_type;     /* rw,ro,sw or xx               */
       int    fs_freq;      /* dump frequency, in days      */
       int    fs_passno;    /* pass number on parallel dump */
       char   *fs_name;     /* name of the file system type */
       char   *fs_opts      /* arbitrary options field      */
};
```

Return Value

A NULL or 0 is returned, but *errno* is not set on detection of errors.

Restrictions

All descriptions are contained in static areas, which should be copied.

Files

/etc/fstab	File system information file.

See Also

fstab(5)

initgroups(3x)

Name

initgroups – initialize group access list

Syntax

initgroups(name, basegid)
char *name;
int basegid;

Description

The `initgroups` subroutine reads through the group file and sets up, using the `setgroups`(2) call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

Restrictions

The `initgroups` subroutine uses the routines based on `getgrent`(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to `initgroups`.

Return Value

The `initgroups` returns –1 if it was not invoked by the superuser.

Files

/etc/group

See Also

setgroups(2)

Name

ldahread – read the archive header of a member of an archive file

Syntax

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

Description

If **TYPE**(*ldptr*) is the archive file magic number, the `ldahread` function reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

The `ldahread` function returns success or failure. If **TYPE**(*ldptr*) does not represent an archive file or if it cannot read the archive header, `ldahread` fails.

See Also

intro(3x), ldclose(3x), ldopen(3x), ar(5), ldfcn(5)

Name

ldclose, ldaclose – close a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

Description

The `ldopen` and `ldclose` functions provide uniform access to simple object files and object files that are members of archive files. An archive of common object files can be processed as if it is a series of simple common object files.

If **TYPE**(*ldptr*) does not represent an archive file, `ldclose` closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE**(*ldptr*) is the magic number for an archive file and if archive has more files, `ldclose` reinitializes **OFFSET**(*ldptr*) to the file address of the next archive member and returns failure. The **LDFILE** structure is prepared for a later `ldopen`(3x). In all other cases, `ldclose` returns success.

The `ldaclose` function closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE**(*ldptr*). The `ldaclose` function always returns success. This function is often used with `ldaopen`.

See Also

fclose(3s), intro(3x) ldopen(3x), ldfcn(5), paths.h(4)

Name

ldfhread – read the file header of a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

Description

The `ldfhread` function reads the file header of the common object file currently associated with *ldptr* . It reads the file header into the area of memory beginning at *filehead*.

The `ldfhread` function returns **success** If *ldfhread* cannot read the file header, it fails.

Usually, `ldfhread` can be avoided by using the macro **HEADER**(*ldptr*) defined in **<ldfcn.h>** see `ldfcn`(5)). Note that the information in HEADER is swapped, if necessary. The information in any field, *fieldname*, of the file header can be accessed using **HEADER**(*ldptr*),*fieldname*.

See Also

intro(3x), ldclose(3x), ldopen(3x), ldfcn(5).

Name

ldgetaux – retrieve an auxiliary entry, given an index

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <sym.h>
#include <ldfcn.h>

pAUXU ldgetaux (ldptr, iaux)
LDFILE ldptr;
long iaux;
```

Description

The `ldgetaux` function returns a pointer to an auxiliary table entry associated with *iaux*. The AUXU is contained in a static buffer. Because the buffer can be overwritten by later calls to `ldgetaux`, it must be copied by the caller if the aux is to be saved or changed.

Note that auxiliary entries are not swapped as this routine cannot detect what manifestation of the AUXU union is retrieved. If LDAUXSWAP(ldptr, ldf) is non-zero, a further call to *swap_aux* is required. Before calling the *swap_aux routine, the caller should copy*

If the auxiliary cannot be retrieved, `ldgetaux` returns null (defined in <stdio.h>) for an object file. This occurs in the following instances:

- The auxiliary table cannot be found
- The *iaux* offset into the auxiliary table is beyond the end of the table

Typically, `ldgetaux` is called immediately after a successful call to `ldtbread` to retrieve the data type information associated with the symbol table entry filled by `ldtbread`. The index field of the symbol, pSYMR, is the *iaux* when data type information is required. If the data type information for a symbol is not present, the index field is *indexNi* and `ldgetaux` should not be called.

See Also

intro(3x), ldclose(3x), ldopen(3x), ldtbseek(3x), ldtbread(3x), ldfcn(5).

Name

ldgetname – retrieve symbol name for object file symbol table entry

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <sym.h>
#include <ldfcn.h>

 char *ldgetname (ldptr, symbol)
LDFILE * ldptr ;
pSYMR * symbol ;
```

Description

The `ldgetname` function returns a pointer to the name associated with *symbol* as a string. The string is contained in a static buffer. Because the buffer can be overwritten by later calls to `ldgetname`, the caller must copy the buffer if the name is to be saved.

If the name cannot be retrieved, `ldgetname` returns null (defined in **<stdio.h>**) for an object file. This occurs in the following instances:

- The string table cannot be found
- The name's offset into the string table is beyond the end of the string table

Typically, `ldgetname` is called immediately after a successful call to `ldtbread`.The `ldgetname` retrieves the name associated with the symbol table entry filled by the function, `ldtbread`.

See Also

intro(3x), ldclose(3x), ldopen(3x), ldtbseek(3x), ldtbread(3x), ldfcn(5).

Name

ldgetpd – retrieve procedure descriptor given a procedure descriptor index

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <sym.h>
#include <ldfcn.h>

long ldgetpd (ldptr, ipd, ppd)
LDFILE ldptr;
long ipd;
pPDR ipd;
```

Description

The `ldgetpd` function returns success or failure depending on whether the procedure descriptor with index *ipd* can be accessed. If it can be accessed, the structure pointed to by *ppd* is filled with the contents of the corresponding procedure descriptor. The *isym, iline,* and *iopt* fields of the procedure descriptor are updated to be used in further LD routine calls. The *adr* field is updated from the symbol referenced by the *isym field.*

The PDR cannot be retrieved when the following occurs:

- The procedure descriptor table cannot be found.
- The ipd offset into the procedure descriptor table is beyond the end of the table.
- The file descriptor that the ipd offset falls into cannot be found.

Typically, `ldgetpd` is called while traversing the table that runs from 0 to SYMHEADER(ldptr).ipdMax – 1.

See Also

ldclose(3x), ldopen(3x), ldtbseek(3x), ldtbread(3x), ldfcn(5)

Name

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldlread (ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINER linent;

int ldlinit (ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem (ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINER linent;
```

Description

The `ldlread` function searches the line number entries of the common object file currently associated with *ldptr*. The `ldlread` function begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, which is the index of its local symbols entry in the object file symbol table. The `ldlread` function reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

The `dlinit` and `ldlitem` functions provide the same behavior as `ldlread`. After an initial call to `ldlread` or `ldlinit`, `ldlitem` can be used to retrieve a series of line number entries associated with a single function. The `ldlinit` function simply finds the line number entries for the function identified by *fcnindx*. The `ldlitem` function finds and reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

The functions `ldlread`, `ldlinit`, and `ldlitem` each return either success or failure. The `ldlread` function fails if one of the following occurs:

- If line number entries do not exist in the object file.
- If *fcnindx* does not index a function entry in the symbol table.
- If it does not find a line number equal to or greater than *linenum*.

The `ldlitem` fails if it does not find a line number equal to or greater than *linenum*.

See Also

ldclose(3x), ldopen(3x), ldtbindex(3x), ldfcn(5)

Name

ldlseek, ldnlseek – seek to line number entries of a section of a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Description

The `ldlseek` function seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The `ldnlseek` function seeks to the line number entries of the section specified by *sectname*.

The `ldlseek` and `ldnlseek` functions return success or failure.

NOTE

Line numbers are not associated with sections in the MIPS symbol table; therefore, the second argument is ignored, but maintained for historical purposes.

If they cannot seek to the specified line number entries, both routines fail.

See Also

ldclose(3x), ldopen(3x), ldshread(3x), ldfcn(5)

Name

ldohseek – seek to the optional file header of a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

Description

The `ldohseek` function seeks to the optional file header of the common object file currently associated with *ldptr*.

`ldohseek` function returns success or failure. If the object file does not have an optional header or if it cannot seek to the optional header, `ldohseek` fails.

The program must be loaded with the object file access routine library **libmld.a**.

See Also

ldclose(3x), ldopen(3x), ldfhread(3x), ldfcn(5)

Name

ldopen, ldaopen – open a common object file for reading

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;

ld readst (ldptr, flags)
LDFILE *ldptr;
intflags;
```

Description

The `ldopen` and `ldclose` functions provide uniform access to simple object files and to object files that are members of archive files. An archive of common object files can be processed as if it is a series of simple common object files.

If *ldptr* has the value null, `ldopen` opens *filename*, allocates and initializes the **LDFILE** structure, and returns a pointer to the structure to the calling program.

If *ldptr* is valid and **TYPE**(*ldptr*) is the archive magic number, `ldopen` reinitializes the **LDFILE** structure for the next archive member of *filename*.

The `ldopen` and `ldclose` functions work in concert. The `ldclose` function returns failure only when only when **TYPE**(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should `ldopen` be called with the current value of *ldptr*. In all other cases, but especially when a new *filename* is opened, `ldopen` should be called with a null *ldptr* argument.

The following is a prototype for the use of `ldopen` and

```
/* for each filename to be processed*/

ldptr = NULL;
do
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )

        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not **NULL**, `ldaopen` opens *filename* anew and allocates and initializes a new **LDFILE** structure, copying the fields from *oldptr*. The `ldaopen` function returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers can be used concurrently to read separate parts of the object file. For example, one pointer can be used to step sequentially through the relocation information while the other is used to read indexed symbol table entries.

The `ldopen` and `ldaopen` functions open *filename* for reading. If *filename* cannot be opened or if memory for the **LDFILE** structure cannot be allocated, both functions return **NULL**. A successful open does not ensure that the given file is a common object file or an archived object file.

The `ldopen` function causes the symbol table header and file descriptor table to be read. Further access, using *ldptr,* causes other appropriate sections of the symbol table to be read (for example, if you call `ldtbread`, the symbols or externals are read). To force sections for each symbol table in memory, call `ldreadst` with *ST_P** constants or'ed together from *st_support.h*.

See Also

fopen(3s), ldclose(3x), ldfcn(5)

Name

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Description

The `ldrseek` function seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The `ldnrseek` function seeks to the relocation entries of the section specified by *sectname*.

The functions `ldrseek` and `ldnrseek` returns success or failure. If *sectindx* is greater than the number of sections in the object file, `ldrseek` fails; if there is no section name corresponding with *sectname,* `ldnrseek` fails. If the specified section does not have relocation entries or if it cannot seek to the specified relocation entries, either function fails.

NOTE

The first section has an index of *one*.

See Also

ldclose(3x), ldopen(3x), ldshread(3x), ldfcn(5)

Name

ldshread, ldnshread – read an indexed or named section header of a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;
```

Description

The `ldshread` function reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

The `ldnshread` functions reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

The `ldshread` and `ldnshread` functions return success or failure. If *sectindx* is greater than the number of sections in the object file, *ldshread* fails. If there is no section name corresponding with *sectname*, `ldnshread` fails. If it cannot read the specified section header, either function fails.

NOTE

The first section header has an index of *one*.

The program must be loaded with the object file access routine library **libmld.a**.

See Also

ldclose(3x), ldopen(3x), ldfcn(5).

Name

ldsseek, ldnsseek – seek to an indexed or named section of a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Description

The `ldsseek` seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The `ldnsseek` seeks to the section specified by *sectname*.

The `ldsseek` and `ldnsseek` return success or failure. If *sectindx* is greater than the number of sections in the object file, `ldsseek` fails; if there is no section name corresponding with *sectname*, `ldnsseek` fails. If a no section data for the specified section does not exist or if it cannot seek to the specified section, either function fails.

NOTE

The first section has an index of *one*.

The program must be loaded with the object file access routine library **libmld.a**.

See Also

ldclose(3x), ldopen(3x), ldshread(3x), ldfcn(5)

Name

ldtbindex – compute the index of a symbol table entry of a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

Description

The `ldtbindex` returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by `ldtbindex` can be used in later calls to `ldtbread`(3x). `ldtbindex` returns the index of the symbol table entry that begins at the current position of the object file; therefore, if `ldtbindex` is called immediately after a particular symbol table entry has been read, it returns the the index of the next entry.

If there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry, `ldtbindex` fails and returns BADINDEX (–1).

Note that the first symbol in the symbol table has an index of *zero*.

See Also

ldclose(3x), ldopen(3x), ldtbread(3x), ldtbseek(3x), ldfcn(5)

Name

ldtbread – read an indexed symbol table entry of a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
pSYMR *symbol;
```

Description

The `ldtbread` reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at *symbol*.

`ldtbread` returns success or failure. If *symindex* is greater than the number of symbols in the object file or if it cannot read the specified symbol table entry, `ldtbread` fails.

The local and external symbols are concatenated into a linear list. Symbols are accessible from symnum zero to *SYMHEADER(ldptr).isymMax+SYMHEADER(ldptr).iextMax*. The index and iss fields of the SYMR are made absolute (rather than file relative) so that routines `ldgetname`(3x), `ldgetaux`(3x), and `ldtbread` proceed normally given those indices. Only the sym part of externals is returned.

Note that the first symbol in the symbol table has an index of *zero*.

See Also

ldclose(3x), ldgetname(3x), ldopen(3x), ldtbseek(3x), ldgetname(3x), ldfcn(5)

Name

ldtbseek – seek to the symbol table of a common object file

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

Description

The `ldtbseek` function seeks to the symbol table of the object file currently associated with *ldptr*.

The `ldtbseek` function returns success or failure. If the symbol table has been stripped from the object file or if it cannot seek to the symbol table, *ldtbseek* fails.

See Also

ldclose(3x), ldopen(3x), ldtbread(3x), ldfcn(5)

Name

malloc, free, realloc, calloc, mallopt, mallinfo – fast main memory allocator

Syntax

```
#include <malloc.h>
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo (max)
int max;
```

Description

The `malloc` and `free` subroutines provide a simple general-purpose memory allocation package, which runs considerably faster than the `malloc`(3) package. It is found in the library `malloc`, and is loaded if the option `-lmalloc` is used with `cc`(1) or `ld`(1).

The `malloc` subroutine returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to `free` is a pointer to a block previously allocated by `malloc`. After `free` is performed, this space is made available for further allocation, and its contents have been destroyed. See `mallopt` below for a way to change this behavior.

Undefined results will occur if the space assigned by `malloc` is overrun or if some random number is handed to `free`.

The `realloc` subroutine changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

The `calloc` subroutine allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

The `mallopt` subroutine provides for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST Set *maxfast* to *value* . The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 0.

M_NLBLKS Set *numlblks* to *value* . The above mentioned large groups each contain

numlblks blocks. The *numlblks* must be greater than 0. The default value for *numlblks* is 100.

M_GRAIN Set *grain* to *value* . The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain* . The *grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set.

M_KEEP Preserve data in a freed block until the next `malloc, realloc,` or `calloc.` This option is provided only for compatibility with the old version of `malloc` and is not recommended.

These values are defined in the malloc.h header file.

The `mallopt` subroutine may be called repeatedly, but may not be called after the first small block is allocated.

The `mallinfo` subroutine provides information describing space usage. It returns the following structure:

```
struct mallinfo  {
        int arena;      /* total space in arena */
        int ordblks;    /* number of ordinary blocks */
        int smblks;     /* number of small blocks */
        int hblkhd;     /* space in holding block headers */
        int hblks;      /* number of holding blocks */
        int usmblks;    /* space in small blocks in use */
        int fsmblks;    /* space in free small blocks */
        int uordblks;   /* space in ordinary blocks in use */
        int fordblks;   /* space in free ordinary blocks */
        int keepcost;   /* space penalty if keep option */
                        /* is used */
}
```

This structure is defined in the malloc.h header file.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

Restrictions

This package usually uses more data space than `malloc`(3).
The code size is also bigger than `malloc`(3).
Note that unlike `malloc`(3), this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of `mallopt` is used.
Undocumented features of `malloc`(3) have not been duplicated.

Return Value

The `malloc`, `realloc`, and `calloc` subroutines return a NULL pointer if there is not enough available memory. When `realloc` returns NULL, the block pointed to by *ptr* is left intact. If `mallopt` is called after any allocation or if *cmd* or *value* are invalid, nonzero is returned. Otherwise, it returns zero.

See Also

brk(2), malloc(3)

Name

nlist – get entries from name list

Syntax

#include <nlist.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];

cc ... -lmld

Description

The `nlist` subroutine examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. For the structure declaration, see */usr/include/nlist.h*.

This subroutine is useful for examining the system name list kept in the file **/vmunix**. In this way programs can obtain system addresses that are up to date.

Diagnostics

If the file cannot be found or if it is not a valid namelist –1 is returned; otherwise, the number of unfound namelist entries is returned.

The type entry is set to 0 if the symbol is not found.

See Also

a.out(5)

Name

openpl, erase, label, line, circle, arc, move, cont, point, linemod, space, closepl, box, color, dot – graphics interface

Syntax

openpl()

erase()

label(*s*)
char *s*[];

line(*x1*, *y1*, *x2*, *y2*)

circle(*x*, *y*, *r*)

arc(*x*, *y*, *x0*, *y0*, *x1*, *y1*)

move(*x*, *y*)

cont(*x*, *y*)

point(*x*, *y*)

linemod(*s*)
char *s*[];

space(*x0*, *y0*, *x1*, *y1*)

closepl()

box(*x0*, *x1*, *y0*, *y1*)

color(*c*)

dot()

Description

These subroutines generate graphic output in a device-independent manner. See `plot`(5) for a description of their effect. The `openpl` subroutine precedes the other subroutines as it opens the device for writing. The `closepl` subroutine flushes the output. The `box, color,` and `dot` routines are used by the lvp16 and hp7475a plotters only.

String arguments to `label` and `linemod` are null-terminated and do not contain newlines.

Many of these functions have additional options for different output devices. They are accessed by the `ld`(1) options as follows:

–lplot	device-independent graphics stream on standard output for `plot`(1g) filters
–lplotaed	AED 512 color graphics terminal
–lplotbg	BBN bitgraph graphics terminal
–lplotdumb	dumb terminals without cursor addressing or line printers
–lplotgigi	gigi graphics terminal

–lplotgrn	grn files
–lplot2648	HP 2648 graphics terminal
–lplot7221	HP 7221 graphics terminal
–lplotimagen	Imagen laser printer (default 240 DPI resolution)
–l300	GSI 300 terminal
–l300s	GSI 300S terminal
–l450	DASI 450 terminal
–l4013	Tektronix 4013 terminal
–l4014	Tektronix 4014 terminal
–llvp16	DEC LVP16 and HP7475A plotters

See Also

graph(1g), plot(1g), plot(5)

Name

ranhashinit, ranhash, ranlookup – access routine for the symbol table definition file in archives

Syntax

```
#include <ar.h>

int ranhashinit(pran, pstr, size)
struct ranlib *pran;
char *pstr;
int size;

ranhash(name)
char *name;

struct ranlib *ranhash(name)
char *name;
```

Description

The function `ranhashinit` initializes static information for future use by `ranhash` and `ranlookup`. The argument *pran* points to an array of ranlib structures. The argument *pstr* points to the corresponding ranlib string table (these are only used by `ranlookup`). The argument size *size* is the size of the hash table and should be a power of 2. If the size is not a power of 2, a 1 is returned; otherwise, a 0 is returned.

The function `ranhash` returns a hash number given a name. It uses a multiplicative hashing algorithm and the *size* argument to `ranhashinit`.

The `ranlookup` function looks up *name* in the ranlib table specified by *ranhashinit*. It uses the `ranhash` routine as a starting point. Then, it does a rehash from there. This routine returns a pointer to a valid ranlib entry on a match. If no matches are found (the "emptiness" can be inferred if the ran_off field is zero), the empty ranlib structure hash table should be sparse. This routine does not expect to run out of places to look in the table. For example, if you collide on all entries in the table, an error is printed tostderr and a zero is returned.

See Also

ar(1), ar(5)

Name

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

Syntax

rem = rcmd(*ahost, inport, locuser, remuser, cmd, fd2p*);
char ***ahost*;
u_short *inport*;
char **locuser, *remuser, *cmd*;
int **fd2p*;

s = rresvport(*port*);
int **port*;

ruserok(*rhost, superuser, ruser, luser*)
char **rhost*;
int *superuser*;
char **ruser, *luser*;

Description

The `rcmd` subroutine is used by the superuser to execute a command on a remote machine using an authentication scheme based on reserved port numbers. The `rresvport` subroutine is a routine that returns a descriptor to a socket with an address in the privileged port space. The `ruserok` subroutine is a routine used by servers to authenticate clients requesting service with `rcmd`. All three functions are present in the same file and are used by the `rshd`(8c) server (among others).

The `rcmd` subroutine looks up the host **ahost* using `gethostbyname`(3n), returning –1 if the host does not exist. For further information, see `gethostent`(3n). Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command as **stdin** and **stdout.** If *fd2p* is nonzero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in `rshd`(8c).

The `rresvport` subroutine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by `rcmd` and several other routines. Privileged addresses consist of a port in the range 0 to 1023. Only the superuser is allowed to bind an address of this sort to a socket.

The `ruserok` subroutine takes a remote host's name, as returned by a `gethostent`(3n) routine, two user names and a flag indicating if the local user's name is the superuser. It then checks the files `/etc/hosts.equiv` and `.rhosts` in the user's home directory to see if the request for service is allowed. A 1 is returned if the machine name is listed in the `hosts.equiv` file, or the host and

remote user name are found in the `.rhosts` file. Otherwise `ruserok` returns –1. If the superuser flag is 1, the checking of the `hosts.equiv` file is bypassed.

See Also

rlogin(1c), rsh(1c), gethostent(3n), rexec(3x), rexecd(8c), rlogind(8c), rshd(8c)

Name

rexec – return stream to a remote command

Syntax

```
rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
u_short inport;
char *user, *passwd, *cmd;
int *fd2p;
```

Description

The `rexec` subroutine looks up the host **ahost* using `gethostbyname`, returning –1 if the host does not exist. For further information, see `gethostent`(3n). Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call ''getservbyname("exec", "tcp")''. For further information, see `getservent`(3n). The protocol for connection is described in detail in `rexecd`(8c).

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command as **stdin** and **stdout**. If *fd2p* is nonzero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

See Also

gethostent(3n), getservent(3n), rcmd(3x), rexecd(8c)

Name

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operation routines

Syntax

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

Description

These functions extract and use capabilities from the terminal capability data base `termcap`(5). These are low level routines; see `curses`(3x) for a higher level package.

The `tgetent` function extracts the entry for terminal *name* into the buffer at *bp*. The *bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to `tgetnum`, `tgetflag`, and `tgetstr`. The `tgetent` function returns –1 if it cannot open the `termcap` file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a pathname rather than `/etc/termcap`. This can speed up entry into programs that call `tgetent`, as well as to help debug new terminal descriptions or to make one for your terminal if you cannot write the file `/etc/termcap`.

The `tgetnum` function gets the numeric value of capability *id*, returning –1 if is not given for the terminal. The `tgetflag` returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. The `tgetstr` function gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes

the abbreviations for this field described in `termcap`(5), except for cursor addressing and padding information.

The `tgoto` function returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables UP (from the **up** capability) and BC (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. Programs that call `tgoto` should be sure to turn off the XTABS bit(s), because `tgoto` may now output a tab. Note that programs using termcap should in general turn off XTABS anyway, because some terminals use control I for other functions, such as nondestructive space. If a **%** sequence is given that is not understood, then `tgoto` returns ''OOPS''.

The `tputs` function decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine that is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by `stty`(3). The external variable PC should contain a pad character to be used (from the **pc** capability) if a null (**^@**) is inappropriate.

Files

/usr/lib/libtermcap.a	–ltermcap library
/etc/termcap	data base

See Also

ex(1), curses(3x), termcap(5)